Refinement to certify abstract interpretations, illustrated on linearization for polyhedra

Sylvain Boulmé, Alexandre Maréchal

Refinement to certify abstract interpretations, illustrated on linearization for polyhedra*

Sylvain Boulmé and Alexandre Maréchal

Univ. Grenoble-Alpes, VERIMAG, F-38000 Grenoble, France
{sylvain.boulme,alex.marechal}@imag.fr

Abstract. Our concern is the modular development of a certified static analyzer in Coq: we extend a certified abstract domain of convex polyhedra with a linearization procedure approximating polynomial expressions. In order to help such a development, we propose a proof framework, embedded in Coq, that implements a refinement calculus. It allows to hide for proofs several low-level aspects of the computations on abstract domains. Moreover, refinement proofs are naturally simplified thanks to computations of weakest preconditions.

1 Introduction

This paper presents two contributions: first, a certified linearization for an abstract domain of convex polyhedra, approximating polynomials by affine constraints ; second, a refinement calculus, helping us to mechanize this proof in Coq [1]. We detail below the context and the features of these two contributions.

1.1 A certified linearization for the abstract domain of polyhedra

We consider the certification of a static analyzer, which aims to ensure absence of undefined behaviors such as division by zero or invalid memory access in an input source program. Such an analyzer computes for each program point an *invariant*: a property that the state at that point must satisfy in all executions. In abstract interpretation [2], invariants are values of datatypes called *abstract domains*. An abstract domain is a syntactic class of properties on memory states. For instance, *convex polyhedra* [3] are conjunctions of affine constraints written $\sum_i a_i x_i \leq b$ where $a_i,\ b \in \mathbb{Q}$ are scalar values and x_i are integer program variables. The abstract domain of convex polyhedra is able to capture relations between program variables (*e.g.* $x + 2 \leq y + x - 2z$). However, it cannot deal directly with non-linear relations, *e.g.* $x^2 - y^2 \leq x \times y$. Thus, linearization techniques are necessary to analyze programs with non-linear arithmetic.

We focus on a linearization technique named intervalization [4], which replaces some variables in a non-linear product by intervals of constants. For instance, if the analysis leads to a state where $x \in [2, 10]$, then guard $x \times y \leq z$

* This work was partially supported by ANR project VERASCO (INS 2011).

can be over-approximated by $[2, 10] \times y \leq z$. The interval is then eliminated by multiplying it with bounds of y or by analyzing the sign of y, leading to an affine constraint usable by the polyhedra domain. More complex and precise linearization methods exist, implying more advanced mathematics such as Bernstein's basis [5] or Handelman representation of polynomials [6]. Intervalization is clearly faster than others [7], and its precision-versus-efficiency trade off may be controlled by several heuristics which are detailed in the paper.

Our certified linearization procedure is now part of the VERIMAG Polyhedra Library (VPL) [8,9], which provides a certified polyhedra domain to VERASCO [10], a certified abstract interpreter for COMPCERT C [11]. Following a design proposed in [12], VPL is organized as a two-tier architecture: an untrusted oracle – combining OCAML and C code – performs most complex computations and outputs a Farkas certificate which is used by a certified front-end to build a correct-by-construction result. As oracles may have side-effects and bugs, they are viewed in COQ as non-deterministic computations of an axiomatized monad [9].

Built on a similar design, our procedure invokes an untrusted oracle that selects certified strategies for linearizing an arithmetic expression. The procedure then checks the validity of the produced certificate and finally computes a correct-by-construction over-approximation of the expression. It is convenient to see such strategies as program transformations, because their correctness is independent from the implementation of the underlying abstract domain and is naturally expressed using concrete semantics of programs [7]. Indeed, a linearization is correct if, in the current context of the analysis, any postcondition satisfied by the output program is also satisfied on the input one (see Figure 1). In such a case, we say that the input program *refines* the output one. This paper aims to explain how refinement helps to develop certified procedures on abstract domains, and in particular our linearization algorithm.

In a context where $x \in [0, 10]$, assignment "$r := x.(y-z)+10.z$" is approximated by the affine program on the right hand-side. Here, operator :∈ performs a non-deterministic assignment.

```
if y − z ≥ 0 then
  r :∈ [10.z, 10.y]
else
  r :∈ [10.y, 10.z]
```

Fig. 1. Intervalization of an assignment involving a sign-analysis

1.2 Certifying computations on abstract domains by refinement

Program refinement consists in decomposing proofs of complex programs by stepwise applications of correctness-preserving transformations. We provide a framework in COQ to apply this methodology for certifying the correctness of computations combining operators of an existing abstract domain. Our framework provides these operators from a Guarded Command Language (GCL) called

$^{\dagger}\mathbb{K}$ and inspired by [13]. A computation $^{\dagger}K$ in $^{\dagger}\mathbb{K}$ comes with two types of semantics: an abstract and a concrete one. Concrete semantics of $^{\dagger}K$ is a transformation on *memory states*. Abstract semantics of $^{\dagger}K$ is a transformation on *abstract states*, *i.e.* on values of the abstract domain. Concrete semantics of $^{\dagger}K$ acts as a *specification* which is *implemented* by its abstract semantics. Indeed, a $^{\dagger}\mathbb{K}$ computation also embeds a proof that abstract semantics is correct *w.r.t.* concrete one: each $^{\dagger}\mathbb{K}$ operator thus preserves correctness by definition. Hence, an OCAML function is extracted from abstract semantics which is certified to be correct *w.r.t.* concrete semantics. In the following, a transformation on abstract (resp. memory) states is called an abstract (resp. concrete) computation.

Hence, taking a piece of code as input, our linearization procedure outputs a $^{\dagger}\mathbb{K}$ computation. The correctness of the procedure is ensured by proving that concrete semantics of its input refines concrete semantics of its output. Informally, it means that the output does not forget any behaviour of the input. Our procedure being developed in a modular way from small intermediate functions, its proof reduces itself to small refinement steps. Each of this refinement step involves only concrete semantics. Our framework provides a tactic simplifying such refinement proofs by computational reflection of weakest-preconditions. The correctness of abstract semantics *w.r.t.* concrete semantics is ensured by construction of $^{\dagger}\mathbb{K}$ operators.

Our framework supports *impure* abstract computations, *i.e.* abstract computations that invokes imperative oracles whose results are *a posteriori* certified. It also allows to reason conveniently about higher-order abstract computations. In particular, our linearization procedure uses a Continuation-Passing-Style (CPS) [14] in order to partition its analyzes according to the sign of affine sub-expressions. For example, in Figure 1, the approximation of the non-linear assignment depends on the sign of $y-z$. In our procedure, CPS is a higher-order programming style which avoids to introduce an explicit datatype handling partitions: this simplifies both the implementation and its proof. This also illustrates the expressive power of our framework, since a simple Hoare logic does not suffice to reason about such higher-order imperative programs.

Our refinement calculus could have applications beyond the correctness of linearization strategies. In particular, the top-level interpreter of the analyzer could also be proved correct in this way. Indeed, the interpreter invokes operations on abstract domains in order to over-approximate any execution of the program, but its correctness does not depend on abstract domains implementations (as soon as these implementations are themselves correct). We illustrate this claim on a toy analyzer, also implemented in COQ.

The mathematics involved in our refinement calculus, relating operational semantics to the lattice structure of monotone predicate transformers, are well-known in abstract interpretation theory [15]. In parallel of our work, the idea to use a refinement calculus in formal proofs of abstract interpreters has been proposed in [16]. Hence, our contribution is more practical than theoretical. On the theoretical side, we propose a refinement calculus dedicated to certification of *impure* abstract computations. On the practical side, we show how to get a concise implementation of such a refinement calculus in Coq and how this helps on a realistic case study: a linearization technique within the abstract interpreter Verasco.

1.3 Overview of the paper

The implementation of our refinement calculus only consists in around 350 lines of Coq (proof scripts included). It is a shallow-embedding of our GCL $^{\dagger}\mathbb{K}$, combining computational reflection of weakest-preconditions [17] with monads [18]. Its understanding requires thus advanced knowledge on Coq, monads, refinement calculus and abstract interpretation. However, the main ideas of our refinement calculus can be understood in a much simpler setting using binary relations instead of monads and weakest-preconditions, and classical set theory instead of Coq.

Section 2 introduces our refinement calculus in this simplified setting, where computations are represented as binary relations. Section 3 presents our certified linearization procedure and how its proof benefits from our refinement calculus. Appendix A explains how we mechanize this refinement calculus in Coq by using smart encodings of binary relations introduced in Section 2. Our Coq sources are available on [19].

2 A refinement calculus for abstract interpretation

We consider an analyzer correct if and only if it rejects all programs that may lead to an *error state*: due to lack of precision, it may also reject safe programs. Section 2.1 defines the notion of error state and semantics of concrete computations. This semantics combines big-steps operational semantics with Hoare Logic. After introducing the notion of abstract computation and its correctness *w.r.t.* a concrete computation, Section 2.2 presents our refinement calculus. Section 2.3 applies our refinement calculus to the certification of higher-order abstract computations.

2.1 Stepwise refinement of concrete computations

The whole paper abusively uses classical set theory, whereas our formalization is in the intuitionistic type theory of Coq without axioms. In particular, it identifies type $A \to$ `Prop` of predicates on A with set $\mathcal{P}(A)$. Hence, we note $\mathcal{R}(A,B) \triangleq \mathcal{P}(A \times B)$ the set of binary relations on $A \times B$. Given R of $\mathcal{R}(A,B)$, we note $x \xrightarrow{R} y$ instead of $(x,y) \in R$. We use operators on $\mathcal{R}(A,A)$ inspired from regular expressions: ε is identity relation on A, $R_1 \cdot R_2$ is "relation R_2 composed with R_1" (i.e. $x \xrightarrow{R_1 \cdot R_2} z \triangleq \exists y, x \xrightarrow{R_1} y \land y \xrightarrow{R_2} z$) and R^* is reflexive and transitive closure of R. We use also notations from the complete lattice structure of $\mathcal{R}(A,B)$. In all the paper, $A \to B$ is a type of *total* functions.

Specifying concrete computations with runtime errors. Given a domain D representing the type of memory states, we add a distinguished element $\lightning$ to D in order to represent the error state: we define $D_{\lightning} \triangleq D \uplus \{\lightning\}$.

We define the set of concrete computations as $\mathbb{K} \triangleq \mathcal{R}(D, D_{\lightning})$. Hence, an element K of $\mathbb{K}$ corresponds to a (possibly) non-deterministic or non-terminating computation from an *input state* of type D to an *output state* of type $D_{\lightning}$. Typically, empty relation represents a computation that loops infinitely for any input. It also represents unreachable code (dead code).

When an error appears, *anything* may happen at runtime. Hence, we introduce $\downarrow K$ that normalizes computation K by returning any output in case of error. It is defined by $d \xrightarrow{\downarrow K} d' \triangleq (d \xrightarrow{K} d' \vee d \xrightarrow{K} \lightning)$.

Refinement pre-order and Hoare specifications. We equip $\mathbb{K}$ with a *refinement* pre-order $\sqsubseteq$ such that $K_1 \sqsubseteq K_2$ iff $K_1 \subseteq \downarrow K_2$ (or equivalently, $\downarrow K_1 \subseteq \downarrow K_2$). Informally, an abstract analysis correct for K_2 is also correct for K_1. The equivalence relation $\equiv$ associated with this pre-order is given by $K_1 \equiv K_2$ iff $\downarrow K_1 = \downarrow K_2$.

Hoare logic is a standard and convenient framework to reason about imperative programs. Let us explain why computations in $\mathbb{K}$ correspond to specifications of Hoare logic. A computation K is equivalently given as a Hoare specification $(\mathfrak{p}_K, \mathfrak{q}_K)$ of $\mathcal{P}(D) \times \mathcal{R}(D, D)$, where $\mathfrak{p}_K$ is a precondition ensuring absence of error, and $\mathfrak{q}_K$ a postcondition relating the initial state to a non-error final state. This equivalence is given by $\mathfrak{p}_K \triangleq D \setminus \{d \mid d \xrightarrow{K} \lightning\}$ and $\mathfrak{q}_K \triangleq K \cap (D \times D)$. The refinement pre-order $K_1 \sqsubseteq K_2$ is equivalent to $\mathfrak{p}_{K_2} \subseteq \mathfrak{p}_{K_1} \wedge \mathfrak{q}_{K_1} \cap (\mathfrak{p}_{K_2} \times D) \subseteq \mathfrak{q}_{K_2}$. Thus, it is equivalent to the usual refinement of specifications in Hoare logic.

Algebra of guarded commands. We now equip $\mathbb{K}$ with an algebra of guarded commands inspired by [13].[1] It combines a *complete* lattice structure with operators inspired from regular expressions. Here, we present this algebra in the case where $\mathbb{K}$ is represented as $\mathcal{R}(D, D_{\lightning})$. In our Coq implementation (given in Appendix A.2), this representation is changed in order to mechanize refinement proofs.

First, the complete lattice structure (for $\sqsubseteq$ pre-order) is given by the usual operators $\cap$ and $\cup$ that we rather note $\sqcap$ and $\sqcup$ in order to keep our notations compatible with the representation-change of Appendix A.2. In our context, $\sqcup$ represents alternatives that may non-deterministically happen at runtime: the analyzer must consider that each of them may happen. Symmetrically, $\sqcap$ represents some choice left to the analyzer.

Empty relation $\emptyset$ is the bottom element and is noted $\perp$. Relation $D \times \{\lightning\}$ is the top element. Given $d \in D_{\lightning}$, we implicitly coerce it as the constant relation $D \times \{d\}$. Hence, the top element of $\mathbb{K}$ lattice is simply noted $\lightning$. Notation $\uparrow f$ explicitly lifts function f of $D \to D$ in $\mathbb{K}$.

[1] However, in our algebra, $\sqsubseteq$ corresponds to "*refines*", whereas in standard refinement calculus it dually corresponds to "*is refined by*". Actually, our convention follows lattice notations of abstract interpretation.

Given a relation $K \in \mathcal{R}(D, D_{\lightning})$, we define its lifting $\upharpoonleft K$ in $\mathcal{R}(D_{\lightning}, D_{\lightning})$ by $\upharpoonleft K \triangleq K \cup \{(\lightning, \lightning)\}$. This allows us to define the sequence of computations by $K_1 \,; K_2 \triangleq K_1 \cdot \upharpoonleft K_2$, and the unbounded iteration of this sequence (*i.e.* a loop with a runtime-chosen number of iterations) by $K^* \triangleq (\upharpoonleft K)^* \cap (D \times D_{\lightning})$.

Given a predicate $P \in \mathcal{P}(D)$, we define $\dashv P \triangleq (P{\times}D) \sqcap \varepsilon$. Informally, if P is satisfied on the current state then $\dashv P$ skips like ε. Otherwise, $\dashv P$ never produces no output like $\bot$. Command $\dashv P$ is called an *assumption* (or a *guard*). We also define the dual notion of *assertion* as $\vdash P \triangleq (\dashv \neg P \,; \lightning) \sqcup \varepsilon$. If P is not satisfied on the current state, then $\vdash P$ produces an error. Otherwise, it skips.

Hence, $\mathbb{K}$ provides a convenient language to express specifications: any Hoare specification (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ is expressed as the computation $\vdash P; Q$. Moreover, refinement allows to express usual Verification Conditions (VC) of Hoare Logic. For our toy analyzer – described later – we use the usual partial correctness VC of unbounded iteration: K^* is equivalent to produce an output satisfying *every* inductive invariant I of K.

$$K^* \equiv \bigsqcap_{I \in \{I \in \mathcal{P}(D) \,\mid\, K \,\sqsubseteq\, \vdash I; D \times I\}} \vdash I; D{\times}I$$

In this equivalence, the $\sqsubseteq$-way corresponds to the soundness of the VC, whereas the $\sqsupseteq$-way corresponds to its completeness. In our context, such a soundness proof typically ensures that the specification of an abstract computation is refined by concrete semantics of the analyzed code. In other words, it ensures that the analysis is correct *w.r.t.* semantics of the analyzed code.

Example on a toy language. Let t stand for an arithmetic term and c be a condition over numerical variables with syntax given in Figure 2. The semantics $[\![t]\!]$ of t and $[\![c]\!]$ of c work with a domain of integer memories $D \triangleq \mathbb{V} \to \mathbb{Z}$ where $\mathbb{V}$ is the type of variables. Hence, $[\![t]\!] \in D \to \mathbb{Z}$ and $[\![c]\!] \in \mathcal{P}(D)$. We omit their definition here.

Let us now introduce a small imperative programming language named $\mathbb{S}$ for which we will describe a toy analyzer in Section 2.2. The syntax of a $\mathbb{S}$ program s is described on Figure 3 together with its big-steps semantics $[\![s]\!]$ defined as an element of $\mathbb{K}$. This semantics is defined recursively on the syntax of s using guarded commands derived from $\mathbb{K}$. First, we define $\dashv c \triangleq \dashv [\![c]\!]$ and $\vdash c \triangleq \vdash [\![c]\!]$. We also use command "$x := t$" defined as $\uparrow \lambda d.d[x := [\![t]\!](d)]$, where memory assignment noted "$d[x := n]$" – for $d \in D$, $x \in \mathbb{V}$ and $n \in \mathbb{Z}$ – is defined as the function $\lambda x' \!:\! \mathbb{V}, \texttt{if}\ x' = x\ \texttt{then}\ n\ \texttt{else}\ d(x')$.

At this point, we have defined an algebra $\mathbb{K}$ of concrete computations: a language that we use to express specifications – for instance, in the form of Hoare specifications – on abstract computations. This algebra also provides denotations for defining big-steps semantics (like in Figure 3). Hence, $\mathbb{K}$ is aimed at providing an intermediate level between operational semantics of programs and their abstract interpretations. Next section defines how we certify correctness of abstract computations *w.r.t.* $\mathbb{K}$ computations.

$$c ::= t_1 \bowtie t_2 \mid \neg c \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \qquad \text{with} \qquad \bowtie \in \{=, \neq, \leq, \geq, <, >\}$$

Fig. 2. Syntax of conditions

s	$\mathbf{assert}(c)$	$x \leftarrow t$	$s_1\,;s_2$	$\mathbf{if}(c)\{s_1\}\mathbf{else}\{s_2\}$	$\mathbf{while}(c)\{s\}$
$[\![s]\!]$	$\vdash c$	$x := t$	$[\![s_1]\!]\,;[\![s_2]\!]$	$\dashv c\,;[\![s_1]\!]$ $\sqcup \dashv\neg c\,;[\![s_2]\!]$	$(\dashv c\,;[\![s]\!])^*\,;\dashv\neg c$

Fig. 3. Syntax and concrete semantics of $\mathbb{S}$

2.2 Certification of abstract computations by diagram composition

Rice's theorem states that the property $d \xrightarrow{K} d'$ is undecidable. In the theory of abstract interpretation, we thus approximate K by a *computable (terminating) function* ${}^\sharp K$ working on an approximation ${}^\sharp D$ of $\mathcal{P}(D)$. Here ${}^\sharp D$ is called an *abstract domain* and its relation to $\mathcal{P}(D)$ is expressed by a concretization function γ of ${}^\sharp D \to \mathcal{P}(D)$. Function ${}^\sharp K$ is called an *abstract interpretation* of K. Here, we say simply that ${}^\sharp K$ is an *abstract computation*.

In this paper, we consider two abstract domains, intervals and convex polyhedra, which are associated with the concrete domain $D \triangleq \mathbb{V} \to \mathbb{Z}$ involved in Figure 3.

1. Given $\mathbb{Z}_\infty \triangleq \mathbb{Z} \uplus \{-\infty, +\infty\}$, an abstract memory ${}^\sharp d$ of interval domain is a finite map associating each variable x with an interval $[a_x, b_x]$ of $\mathbb{Z}_\infty^2$. Its concretization is the set of concrete memory states satisfying the constraints of ${}^\sharp d$, *i.e.* $\gamma({}^\sharp d) \triangleq \{d \in D \,|\, \forall x, a_x \leq d(x) \leq b_x\}$.
2. The concretization of a convex polyhedron ${}^\sharp d = \bigwedge_i \sum_j a_{ij}.x_j \leq b_i$, where a_{ij} and b_i are rational constants, and x_j are integer program variables is $\gamma({}^\sharp d) \triangleq \{d \in D \mid \bigwedge_i \sum_j a_{ij}.d(x_j) \leq b_i\}$.

Correctness diagram of impure abstract computations. In our framework, we do not prove that abstract computations terminate: we only prove that they are a sound over-approximation of their corresponding concrete computation. Moreover, abstract computations may invoke untrusted oracles, whose results are verified by a certified checker. But, a bug in those oracles may make the whole computation non-deterministic or divergent. Thus, it is potentially unsound to consider abstract computations as pure functions. In this simplified presentation of our framework, we define abstract computations as relations in $\mathcal{R}({}^\sharp D, {}^\sharp D)$. In order to extract abstract computations from Coq to Ocaml functions, we improve this representation of abstract computations in Appendix A.1.

We express correctness of abstract computations through commutative diagrams represented on the right hand side and defined as follows.

Definition (correctness of abstract computations). *An abstract computation* $^\sharp K \in \mathcal{R}(^\sharp D, {^\sharp D})$ *is correct w.r.t. a concrete computation* $K \in \mathcal{R}(D, D_\lightning)$*, iff*

$$\forall {^\sharp d}, {^\sharp d'} \in {^\sharp D} \quad \forall d \in D, \forall d' \in D_\lightning,$$

$$^\sharp d \xrightarrow{^\sharp K} {^\sharp d'} \ \wedge\ d \xrightarrow{K} d' \ \wedge\ d \in \gamma(^\sharp d) \qquad \Rightarrow \qquad d' \in \gamma(^\sharp d')$$

$$\begin{tikzcd}
^\sharp d \arrow[r, "^\sharp K"] \arrow[d, "\gamma"'] & ^\sharp d' \arrow[d, dotted, "\gamma"] \\
d \arrow[r, "K"'] & d'
\end{tikzcd}$$

Note that $d' \in \gamma(^\sharp d')$ implies itself that $d' \neq \lightning$.

Such a diagram thus corresponds to a pair of an abstract computation and a concrete computation, with a proof that the abstract one is correct *w.r.t.* the concrete one. As illustrated on the example below, these diagrams allow to build *compositional proofs* that an abstract computation, composed of several simpler parts, is correct *w.r.t.* a concrete computation. Diagrams are indeed preserved by several composition operators, and also by refinement of concrete computations.

As example, consider two abstract computations $^\sharp K_1$ and $^\sharp K_2$ which are correct *w.r.t.* concrete K_1 and K_2. In order to show that abstract computation $^\sharp K_1 \cdot {^\sharp K_2}$ is correct *w.r.t.* concrete K, it suffices to prove that $K \sqsubseteq K_1 \,;K_2$, as illustrated on the right hand side scheme.

$$\begin{tikzcd}
{} \arrow[r, "^\sharp K_1"] \arrow[d, "\gamma"'] & {} \arrow[r, "^\sharp K_2"] \arrow[d, dotted, "\gamma"] & {} \arrow[d, dotted, "\gamma"] \\
{} \arrow[r, dotted, "K_1"'] \arrow[d, "="'] & {} \arrow[r, dotted, "K_2"'] \arrow[d, phantom, "K \sqsubseteq K_1\,; K_2"] & {} \arrow[d, dotted, "="] \\
{} \arrow[rr, "K"'] & {} & {}
\end{tikzcd}$$

In the following, we introduce a datatype noted $^\dagger\mathbb{K}$ to represent these diagrams: a diagram $^\dagger K \in {^\dagger\mathbb{K}}$ represents an abstract computation $^\sharp K$ which is correct *w.r.t.* its associated concrete computation K. The core of our approach is to lift guarded-commands on $\mathbb{K}$ involved in Figure 3 as guarded-commands on $^\dagger\mathbb{K}$. For instance, our toy analyzer $^\sharp[\![s]\!]$ for s in $\mathbb{S}$ is defined similarly to $[\![s]\!]$ of Figure 3, but from $^\dagger\mathbb{K}$ operators instead of $\mathbb{K}$. For a given diagram $^\dagger K$, we can prove the correctness of abstract computation $^\sharp K$ *w.r.t.* a concrete computation K' simply by proving that $K' \sqsubseteq K$. In practice, such refinement proofs are simplified using a weakest-liberal-precondition calculus (see Appendix A.2).

VPL *interface of abstract domains.* We derive our guarded-commands on $^\dagger\mathbb{K}$ in a generic way from the VPL interface of abstract domains, introduced in [9] and reformulated here on Figure 4. This interface differs from Verasco's one because it allows impure operators.[2] Besides its concretization function γ, a VPL abstract domain $^\sharp D$ provides constants $^\sharp\top$ and $^\sharp\bot$, representing respectively predicate true and false. It also provides abstract computations $^\sharp\!\dashv c$ and $x \overset{\sharp}{:=} t$ of $\mathcal{R}(^\sharp D, {^\sharp D})$, which are respectively correct *w.r.t.* concrete computations $\dashv c$ and $x := t$. It provides operator $^\sharp\sqcup$ of $\mathcal{R}(^\sharp D {\times} {^\sharp D}, {^\sharp D})$, which over-approximates binary union on $\mathcal{P}(D)$. At last, it provides inclusion test $^\sharp\sqsubseteq$ of $\mathcal{R}(^\sharp D {\times} {^\sharp D}, \mathtt{bool})$.

[2] Coercing a VPL abstract domain into a Verasco one thus remains to assume that the underlying oracles are observationally pure, see [9].

$$D \subseteq \gamma(^\sharp\top) \qquad \gamma(^\sharp\bot) \subseteq \emptyset \qquad {}^\sharp d \xrightarrow{^\sharp\dashv c} {}^\sharp d' \;\Rightarrow\; \gamma(^\sharp d) \cap [\![c]\!] \subseteq \gamma(^\sharp d')$$

$${}^\sharp d \xrightarrow{x \overset{\sharp}{:=} t} {}^\sharp d' \wedge d \in \gamma(^\sharp d) \Rightarrow d[x := [\![t]\!](d)] \in \gamma(^\sharp d')$$

$$(^\sharp d_1, {}^\sharp d_2) \overset{^\sharp\sqcup}{\rightarrow} {}^\sharp d' \Rightarrow \gamma(^\sharp d_1) \cup \gamma(^\sharp d_2) \subseteq \gamma(^\sharp d') \qquad (^\sharp d_1, {}^\sharp d_2) \overset{^\sharp\sqsubseteq}{\rightarrow} \texttt{true} \Rightarrow \gamma(^\sharp d_1) \subseteq \gamma(^\sharp d_2)$$

Fig. 4. Correctness specifications of VPL abstract domains

Abstract computations of guarded-commands. We now lift each $\mathbb{K}$ guarded-command of Figure 3 into a $^\dagger\mathbb{K}$ guarded-command. Each $^\dagger\mathbb{K}$ operator has the same notation than its corresponding $\mathbb{K}$ operator. Below, we associate each concrete operator of Figure 3 with an abstract computation. The diagrammatic proof relating them is straightforward from correctness specifications given at Figure 4.

Concrete commands $\dashv c$ and $x := t$ are associated with $^\sharp\dashv c$ and $x \overset{\sharp}{:=} t$. Concrete command $K_1\,; K_2$ is associated with $^\sharp K_1 \cdot \underline{^\sharp K_2}$ – where $\underline{^\sharp K_2}$ returns $^\sharp\bot$ if the current abstract state is included in $^\sharp\bot$, or else runs $^\sharp K_2$. Concrete $K_1 \sqcup K_2$ is lifted by applying operator $^\sharp\sqcup$ to the results of $^\sharp K_1$ and $^\sharp K_2$.

Concrete assertion $\vdash c$ is associated with checking that the result of $^\sharp\dashv\neg c$ is *included* in $^\sharp\bot$: otherwise, the abstract computation fails (in practice, it may raise an alarm for the user, see Section A.1). Hence, concrete $\lightning$ is associated with abstract computation $\emptyset$ (concrete $\bot$ is associated with $^\sharp\bot$).

At last, concrete K^* is associated with an abstract computation which invokes an untrusted oracle proposing an inductive invariant of $^\sharp K$ for the current abstract state. Thus, using inclusion tests, $^\sharp(K^*)$ checks that the invariant proposed by the oracle is actually an inductive invariant (otherwise, it fails).

2.3 Higher-order programming with correctness diagrams

Our linearization procedure detailed in Section 3.2 illustrates how we use GCL $^\dagger\mathbb{K}$ as a programming language for abstract computations. GCL $\mathbb{K}$ is our specification language. Each program $^\dagger K$ of $^\dagger\mathbb{K}$ is associated with a specification K of $\mathbb{K}$ syntactically derived from its code meaning that each $^\dagger\mathbb{K}$ operator is syntactically associated with the $\mathbb{K}$ operator from which it is lifted in the above paragraph.

Our linearization procedure invokes two other operators of $^\dagger\mathbb{K}$. First, an operator which *casts* $^\dagger K$ to a given specification K': it requires $K' \sqsubseteq K$ in order to produce a new valid $^\dagger\mathbb{K}$ diagram. This cast operator thus leads to a modular design of the certified development since it allows stepwise refinement of $^\dagger\mathbb{K}$ diagrams. Second, given a computation π of $\mathcal{R}(^\sharp D, A)$ where A is a given type, it invokes an operator binding results of π to a function $^\dagger g$ of $A \rightarrow {}^\dagger\mathbb{K}$. This operator requires a *concrete* postcondition Q of $A \rightarrow \mathcal{P}(D)$ on the results of π. In other words, under the condition $\forall^\sharp d, \forall x \in A, {}^\sharp d \overset{\pi}{\rightarrow} x \Rightarrow \gamma(^\sharp d) \subseteq Q\,x$, we define diagram $\pi {}^{\dagger}\!\!\gg\!\!=_{Q} {}^{\dagger}g$ as abstract computation $\{({}^{\sharp}d_1, {}^{\sharp}d_2) \mid \exists x, {}^{\sharp}d_1 \overset{\pi}{\to} x \wedge {}^{\sharp}d_1 \xrightarrow{{}^{\sharp}g\,x} {}^{\sharp}d_2\}$ specified by $\bigsqcap_x \vdash Q\,x\,;g\,x$.

Actually, Section 3.2 applies our refinement calculus to certify higher-order abstract computations. Indeed, our linearization procedure *partitions* abstract states in order to increase precision. Continuation-Passing-Style (CPS) [14] is a higher-order pattern which provides a lightweight and modular style to program and certify simple partitioning strategies. Let us now detail this idea.

Typically, given an abstract state ${}^{\sharp}d$, our linearization procedure invokes a sub-procedure ${}^{\sharp}f$ that splits ${}^{\sharp}d$ into a partition $({}^{\sharp}d_i)_{i\in I}$ and computes a value r_i (of a given type A) for each cell ${}^{\sharp}d_i$. Then, the linearization procedure *continues* the computation from $(r_i, {}^{\sharp}d_i)$ in each cell to finally return the join of all cells. In other words, from ${}^{\sharp}d$, ${}^{\sharp}f$ computes $(r_i, {}^{\sharp}d_i)_{i\in I}$. The main procedure finally computes ${}^{\sharp}\bigsqcup_{i\in I}({}^{\sharp}g\; r_i\; {}^{\sharp}d_i)$ – where ${}^{\sharp}g$ is a given function of $A \to \mathcal{R}({}^{\sharp}D, {}^{\sharp}D)$. In order to avoid explicit handling of partitions, we make ${}^{\sharp}g$ a parameter of ${}^{\sharp}f$ to perform the join inside ${}^{\sharp}f$. In this style, ${}^{\sharp}f$ is of type $(A \to \mathcal{R}({}^{\sharp}D, {}^{\sharp}D)) \to \mathcal{R}({}^{\sharp}D, {}^{\sharp}D)$ and the parameter ${}^{\sharp}g$ of ${}^{\sharp}f$ is called its *continuation*.

However, specifying directly the correctness of such abstract computations is not obvious because of the higher-order parameter. Actually, we define ${}^{\dagger}f$ of type $(A \to {}^{\dagger}\mathbb{K}) \to {}^{\dagger}\mathbb{K}$ and work with a continuation ${}^{\dagger}g$ of type $A \to {}^{\dagger}\mathbb{K}$. As we shall see later, this trick keeps implicit the notion of partition, both in specification and in implementation.

Similarly, CPS allows to implement some trace-partitioning without requiring a trace-partitioning domain[20]. Trace-partitioning is used in abstract interpretation because we have $({}^{\sharp}K_1 \cdot {}^{\sharp}K_3) {}^{\sharp}\!\sqcup ({}^{\sharp}K_2 \cdot {}^{\sharp}K_3)\ {}^{\sharp}\!\sqsubseteq\ ({}^{\sharp}K_1 {}^{\sharp}\!\sqcup {}^{\sharp}K_2) \cdot {}^{\sharp}K_3$, but the converse is not true. Hence, the left side is more precise whereas the right is faster (computation ${}^{\sharp}K_3$ is factorized). In practice, trace-partitioning strategies select the left or the right side according to information of the current abstract state. Using CPS, we define ${}^{\dagger}f \triangleq \lambda {}^{\dagger}g, ({}^{\dagger}g\; {}^{\dagger}K_1) \sqcup ({}^{\dagger}g\; {}^{\dagger}K_2)$. Then, the left side derives from ${}^{\dagger}f\; \lambda {}^{\dagger}K, ({}^{\dagger}K\,; {}^{\dagger}K_3)$ whereas the right side derives from $({}^{\dagger}f\; \lambda {}^{\dagger}K, {}^{\dagger}K)\,; {}^{\dagger}K_3$.

3 Interval-based linearization strategies for polyhedra

As described in [9], VPL works with affine terms given by the abstract syntax $t ::= n \mid x \mid t_1 + t_2 \mid n.t$ where x is a variable, n a constant of $\mathbb{Z}$. We now extend VPL operators of Figure 4 to support polynomial terms, where "$n.t$" product is generalized into "$t_1 \times t_2$".

VPL derives assignment operator ${}^{\sharp}\!\!:=$ from guard ${}^{\sharp}\!\!\dashv$ and low-level operators which do not involve terms but only variables: projection and renaming. It also derives guard operator from a restricted one where conditions have the form $0 \bowtie t$ where $\bowtie\, \in \{\leq, =, \neq\}$. Hence, we focus on the linearization of the restricted guard ${}^{\sharp}\!\!\dashv 0 \bowtie t$, where t is a polynomial. In the following, we use letter p for polynomials and only keep letter t for affine terms.

Roughly speaking, we approximate a guard ${}^{\sharp}\!\!\dashv 0 \bowtie p$ by guards ${}^{\sharp}\!\!\dashv 0 \bowtie [t_1, t_2]$ – where t_1 and t_2 are affine terms or infinite bounds – such that, in the current abstract state, p is included in affine interval $[t_1, t_2]$. Approximated guards

$^\sharp\!\dashv 0 \bowtie [t_1, t_2]$ are defined by cases on $\bowtie$:

$\bowtie$	$\leq$	$=$	$\neq$
$^\sharp\!\dashv 0 \bowtie [t_1, t_2]$	$^\sharp\!\dashv 0 \leq t_2$	$^\sharp\!\dashv 0 \leq t_2 \wedge t_1 \leq 0$	$^\sharp\!\dashv 0 < t_2 \vee t_1 < 0$

Affine intervals are computed using heuristics inspired from [4], except that in order to increase precision, we dynamically split the abstract state $^\sharp d$ into a partition $(^\sharp d_i)_{i \in I}$ according to the sign of some affine subterms. Hence, each cell $^\sharp d_i$ may lead to a distinct affine interval $[t_{i,1}, t_{i,2}]$. Finally, $^\sharp\!\dashv 0 \bowtie p$ is over-approximated by computing the join of all $^\sharp\!\dashv 0 \bowtie [t_{i,1}, t_{i,2}]$ in each cell $^\sharp d_i$. For instance, consider a polyhedron $^\sharp d$ such that $^\sharp d \subseteq \{\, (x, y, z) \in \mathbb{Z}^3 \mid -2 \leq x \leq 5 \,\}$ and the guard $^\sharp\!\dashv 0 \leq x \times (y + x) + z$. We choose to intervalize the first occurrence of x by the interval $[-2, 5]$, directly derived from $^\sharp d$. Hence, we partition $^\sharp d$ on the sign of the sub-expression $(y + x)$, leading us to interval $[-2(y + x), 5(y + x)] + z$ in cell $^\sharp d \cap \{\, (x, y, z) \in \mathbb{Z}^3 \mid 0 \leq y + x \,\}$ and $[5(y + x), -2(y + x)] + z$ in cell $^\sharp d \cap \{\, (x, y, z) \in \mathbb{Z}^3 \mid y + x \leq -1 \,\}$. Hence, with this strategy $^\sharp\!\dashv 0 \leq x \times (y + x) + z$ is computed as

$$\big(^\sharp\!\dashv 0 \leq y + x \cdot {}^\sharp\!\dashv 0 \leq 5(y + x) + z\big) \,^\sharp\!\sqcup \big(^\sharp\!\dashv y + x \leq -1 \cdot {}^\sharp\!\dashv 0 \leq -2(y + x) + z\big)$$

Our certified linearization is built on a two-tier architecture: an untrusted oracle uses heuristics to select linearization strategies and a certified procedure applies them to build a correct-by-construction result. Section 3.1 lists these strategies and their effect on the precision-versus-efficiency trade-off. Section 3.2 details the design of our certified procedure and of our oracle. It also illustrates our lightweight handling of partitions using CPS in our certified procedure.

3.1 Our list of interval-based strategies

Constant Intervalization. Our fastest strategy applies an intervalization operator of the abstract domain. Given a polynomial p, this operator, written $^\sharp\pi(p)$, over-approximates p by an interval where affine terms are reduced to constants. More formally, $^\sharp\pi(p)$ is a computation of $\mathcal{R}(^\sharp D, \mathbb{Z}^2_\infty)$ satisfying $^\sharp d \xrightarrow{^\sharp\pi(p)} [n_1, n_2] \Rightarrow \gamma(^\sharp d) \subseteq \{d \mid n_1 \leq [\![p]\!]d \leq n_2\}$. It uses a naive interval domain, built on the top of polyhedral domain. Arithmetic operations $+$ and $\times$ are approximated by corresponding operations on intervals:

$$[n_1, n_2] + [n_3, n_4] \triangleq [n_1 + n_3, n_2 + n_4]$$

$$[n_1, n_2] \times [n_3, n_4] \triangleq [\min(E), \max(E)] \text{ where } E = \{n_1.n_3, n_1.n_4, n_2.n_3, n_2.n_4\}$$

Ring rewriting. A major weakness of $^\sharp\pi$ operator is its sensitivity to ring rewritings. For instance, consider a polynomial p such that $^\sharp\pi(p)$ returns $[0, n]$ where $n \in \mathbb{N}^+$. Then $^\sharp\pi(p - p)$ returns $[-n, n]$ instead of the precise result 0. Such imprecision occurs in barycentric computations such as $p_1 \triangleq p \times t_1 + (n - p) \times t_2$ where affine terms t_1, t_2 are bounded by $[n_1, n_2]$. Indeed $^\sharp\pi(p_1)$ returns $2n.[n_1, n_2]$ instead of a more precise $n.[n_1, n_2]$. Moreover, if we rewrite p_1 into an equivalent polynomial $p_2 \triangleq p \times (t_1 - t_2) + n.t_2$, then $^\sharp\pi(p_2)$ returns $n.[2.n_1 - n_2, 2.n_2 - n_1]$. If

$n_1 > 0$ or $n_2 < 0$, then $^\sharp\pi(p_2)$ is strictly more precise than $^\sharp\pi(p_1)$. But, if $n_1 < 0$ and $n_2 > 0$, then the situation is reversed.

That's the reason why our oracle begins by simplifying the polynomial before trying to factorize it conveniently. But as illustrated above, it is difficult to find a factorization minimizing $^\sharp\pi$ results. We give more details on the ring rewriting heuristics of our oracle in the following.

Sign partitioning. Given a polynomial p' and an affine term t, partitioning according to the sign of t gives more precise bounds of $p' \times t$ than $^\sharp\pi(p' \times t)$. Indeed, assuming that $^\sharp\pi(p')$ returns $[n'_1, n'_2]$, if $0 \leq t$, then $p' \times t \in [n'_1.t, n'_2.t]$, otherwise $p' \times t \in [n'_2.t, n'_1.t]$. When the sign of t is known, sign partitioning allows to discard one of these two cases and thus gives a fast affine approximation of $p' \times t$. Its main drawback is a worst-case exponential blow-up if applied systematically.

Let us illustrate sign partitioning for the previous barycentric-like computation of p_2. As said above, $p_2 \in [n.t_2, n.t_1]$ in cell $0 \leq t_1 - t_2$, whereas $p_2 \in [n.t_2, n.t_1]$ in cell $t_1 - t_2 < 0$. When we join the two cells, p_2 is bounded by constant interval $n.[n_1, n_2]$ as expected for such a barycentre. Let us remark that sign partitioning is also sensitive to ring rewritings: for p_1, it does not give such a precise bound.

Another issue with sign partitioning comes from the selection of the affine term involved in the sign analysis. For instance, on a product of affine terms $t_1 \times t_2$, sign partitioning discards at least one of the affine terms in the resulting approximation. Again, selecting the best choice is difficult. By convention, our certified procedure partitions the sign of the right affine term. Hence, for the extern oracle, selecting the affine term under partitioning is a particular case of ring rewriting (*e.g.* rewriting $t_1 \times t_2$ into $t_2 \times t_1$).

Focusing. Focusing is a ring rewriting heuristic which may increase the precision of sign partitioning. Given a product $p \triangleq t_1 \times t_2$, the *focusing* of t_2 in center n consists in rewriting p into $p' \triangleq n.t_1 + t_1 \times (t_2 - n)$. Thanks to this focusing, the affine term $n.t_1$ appears whereas t_1 would otherwise be discarded by sign partitioning. Let us simply illustrate the effect of this rewriting when $0 \leq n \leq n'_1$ with t_1 (resp. t_2) bounded by $[n_1, n_2]$ (resp. $[n'_1, n'_2]$). Sign partitioning bounds p in affine interval $[n_1.t_2, n_2.t_2]$ whereas p' is bounded by interval $[n_1.t_2 + n.(t_1 - n_1), n_2.t_2 - n.(n_2 - t_1)]$. The former contains the latter since $t_1 - n_1$ and $n_2 - t_1$ are non-negative. Under these assumptions, the precision is maximal when $n = n'_1$.

Applied carelessly, focusing may also decrease the precision. That's why on products $p \times t$ where t is bounded by $[n_1, n_2]$, our oracle uses the following heuristic which can not decrease precision: if $0 \leq n_1$, then focus t in center n_1; if $n_2 \leq 0$, then focus t in center n_2; otherwise, do not try to change focus of t.

Static vs dynamic intervalization during partitioning. Computing the constant bounds of an affine term (even a variable) inside a given polyhedron invokes costly linear programming procedures. Hence, for a given polynomial p to approximate, we start by computing an environment σ that associates each variable of p with a constant interval. This environment indeed helps the oracle to select a strategy. Moreover, to avoid several runs of similar linear programming problems, $^\sharp\pi$ operator should be invoked in *static* mode, instead of the default *dynamic* mode. In static mode, $^\sharp\pi$ intervalizes from σ using only interval arithmetic: this is the fastest but less precise mode. In dynamic mode, $^\sharp\pi$ intervalizes affine subterms in the current cell (generated from sign partitioning) using linear programming. In this mode, interval arithmetic is only used on non-linear subterms.

For instance, let us consider the sign-partitioning of $p \triangleq t_1 \times t_2$ in the context $0 < n_1, n_2$ and $-n_1 \leq t_2 \leq t_1 \leq n_2$. In cell $0 \leq t_2$, static mode bounds p by $[-n_1.t_2, n_2.t_2]$ whereas, dynamic mode bounds p by $[0, n_2.t_2]$. In cell $t_2 < 0$, both mode bounds p by $[n_2.t_2, -n_1.t_2]$. On the join of these cells, both mode gives the same upper bound. But lower bound is $-n_1.n_2$ for static mode, whereas it is $\frac{n_1.n_2}{n_1+n_2}(t_2 + n_1) - n_1.n_2$ for dynamic mode, which is strictly more precise.

3.2 Design of our implementation

For a guard $^\sharp\dashv 0 \bowtie p$, our certified procedure first rewrites p into $p' + t$ where t is an affine term and p' a polynomial. This may keep the non-affine part p' small compared with the affine one t. Typically, if p' is syntactically equal to zero, we simply apply the standard affine guard $^\sharp\dashv 0 \bowtie t$. Otherwise, we compute environment σ for variable of p' and invoke our extern oracle on p' and σ. This oracle returns a polynomial p'' enriched with tags on subexpressions. We handle three tags: `AFFINE` expresses that the subexpression is affine; `STATIC` expresses that the subexpression has to be intervalized in static mode; `INTERV` expresses that intervalization is done using only $^\sharp\pi$ (instead of sign-partitioning). From the result p'' of the oracle, our certified procedure checks that $p' = p''$ using a normalization procedure for polynomials defined in the standard distribution of Coq (see [21]). If this is not the case, our procedure simply skips (or raises an error according to a compile-time option). Otherwise, it invokes a CPS affine intervalization of p'' for continuation $\lambda[t_1, t_2], \dashv 0 \bowtie [t_1 + t, t_2 + t]$. The next paragraphs detail this certified CPS affine intervalization and then, our extern oracle.

Certified CPS affine intervalization. We implement and prove our affine intervalization using the CPS technique described in Section 2.3. On polynomial p'' and continuation $^\dagger g$, the specification of our CPS intervalization is

$$\varepsilon \sqcap \bigsqcap_{[t_1,t_2]} \vdash \{d \mid t_1 \leq [\![p'']\!]d \leq t_2]\} \,;\, g[t_1, t_2]$$

The ε case corresponds to a failure of our procedure: typically, a subexpression is not affine as claimed by the extern oracle. In case of success, the procedure selects non-deterministically some affine intervals $[t_1, t_2]$ bounding p'' before merging continuations on them. The procedure is implemented recursively over the syntax of p''. Figure 5 sketches the implementation and the specification of the sign-partitioning subprocedure. This latter deals with a particular case where p'' is

Given ${}^{\dagger}\pi\, p$ of $(\mathbb{Z}^2_\infty \to {}^{\dagger}\mathbb{K}) \to {}^{\dagger}\mathbb{K}$ defined by ${}^{\dagger}\pi\, p\, {}^{\dagger}g_0 \triangleq {}^{\sharp}\pi(p)\; {}^{\dagger}\!\!\gg\!\!=_{\lambda[n_1,n_2],\{d \mid n_1 \leq [\![p]\!]d \leq n_2\}} {}^{\dagger}g_0$ the ${}^{\dagger}\mathbb{K}$ program on the right-hand side satisfies the specification below:

$$\bigsqcap_{[t_1,t_2]} \vdash \{d \mid t_1 \leq [\![p \times t]\!]d \leq t_2\}\,;\, g[t_1,t_2]$$

$$\boxed{\begin{array}{l}
\textbf{if } \text{static } \textbf{then} \\
\quad {}^{\dagger}\pi\ p\ \ (\lambda[n_1,n_2],\ (\dashv 0 \leq t\,;\, {}^{\dagger}g[n_1.t, n_2.t]) \\
\qquad\qquad\qquad\qquad \sqcup\ (\dashv t < 0\,;\, {}^{\dagger}g[n_2.t, n_1.t])) \\
\textbf{else} \\
\qquad (\dashv 0 \leq t\,;\, {}^{\dagger}\pi\ p\ \lambda[n_1,n_2],\ {}^{\dagger}g[n_1.t, n_2.t]) \\
\quad \sqcup\ (\dashv t < 0\,;\, {}^{\dagger}\pi\ p\ \lambda[n_1,n_2],\ {}^{\dagger}g[n_2.t, n_1.t])
\end{array}}$$

Fig. 5. Sign-partitioning for $p \times t$ with continuation ${}^{\dagger}g$

a polynomial written $p \times t$ with t affine. In the implementation part, boolean `static` indicates the mode of ${}^{\sharp}\pi$. In static mode, we indeed factorize computation of ${}^{\sharp}\pi$ on both cells of the partition.

Our linearization procedure is written in around 2000 Coq lines (proofs included). Among them, the CPS procedure and its subprocedures take only 200 lines. The bigger part – around 1000 lines – is thus taken by arithmetic operators on interval domains (constant and affine intervals).

Design of our extern oracle. Only fast strategies may be tractable on big polynomials. Hence, our extern oracle may select systematically static constant intervalization on big polynomials. Otherwise, it ranks variables according to their priority to be discarded by sign-partitioning. Then, it factorizes variables with the highest priority. The priority rank is mainly computed from the size of intervals in the precomputed environment σ: unbounded variables must not be discarded whereas variables bounded by a singleton are always discarded by static intervalization. Our oracle also tries to minimize the number of distinct variables that are discarded: hence, variables appearing in many monomials have a higher priority. The oracle also interleaves factorization with focusing. Our oracle is written in 1300 lines of Ocaml code.

4 Conclusion & Perspectives

We have extended the VPL with certified handling of non-linear multiplication by a modular and novel design. Our computations are performed by an untrusted oracle which delivers a certificate to a certified front-end. Our proofs use diagrammatic constructs based on stepwise refinement calculus. To improve the over-approximation precision, the abstract domain is partitioned implicitly thanks to a Continuation-Passing-Style design. Refinement proofs are finally simplified by the computations of Weakest-Liberal-Preconditions, making them clear and concise.

Our linearization procedures is able to give a fast over-approximation of polynomials thanks to variable intervalization. Precision is increased thanks to domain partitioning and dynamic computation of bounding affine terms. Hence, we may finely tune the precision-versus-efficiency tradeoff in our oracle. More sophisticated linearization methods such as Bernstein polynomials or Handelman representation offer the guaranty to converge, meaning that their result get more precise as we let them time. However, they currently require heavy computation costs that, added with the already expensive polyhedral operators, seem too massive to be exploited in VERASCO. Such linearizations must first be algorithmically refined before being usable in abstract interpreters.

Because floating arithmetic would make us explicitly handle error terms at each operation, the VPL is for now limited to integers, as well as our linearization. Our implementation also lacks other operators such as division or modulo. For these reasons, it is hard to evaluate our method on real-life programs. Currently, our tests are limited to small handmade examples focusing on classes of mathematical problems, such as parabola or barycentric approximations. On these cases, our oracle is able to give much more precise approximations than VERASCO interval domain.

Figure 3 sketches how we certified a toy analyzer from a simple big-steps semantics: we simply interpret the operators of concrete semantics in abstract semantics. Appendix A.1 details how this toy analyzer handles alarms in the style of VERASCO. We may wonder whether our approach scales up on a complex language like COMPCERT C. In particular, COMPCERT semantics is not a simple big-steps one. For instance, it distinguishes between a diverging program that do not invoke any system call and a diverging program that invokes some. Such a distinction cannot be done by our concrete semantics. However, such a feature does not seem necessary to the correctness of VERASCO analysis, since such programs do not have any undefined behavior. Hence, our approach would introduce an abstraction over COMPCERT semantics which should even ease the proof of the analyzer.

References

1. The Coq Development Team: The Coq proof assistant reference manual – version 8.4. INRIA. (2012-2014)
2. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM (1977)
3. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, ACM (1978)
4. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI. Volume 3855 of LNCS., Springer (2006)
5. Farouki, R.T.: The Bernstein polynomial basis: A centennial retrospective. Computer Aided Geometric Design **29**(6) (2012)
6. Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. Pacific Journal of Mathematics **132**(1) (1988)
7. Maréchal, A., Périn, M.: Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra. Technical Report TR-2014-7, Verimag Research Report (july 2014)
8. Fouilhé, A., Monniaux, D., Périn, M.: Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In: SAS. Volume 7935., Springer (2013)

9. Fouilhé, A., Boulmé, S.: A certifying frontend for (sub)polyhedral abstract domains. In: VSTTE. Volume 8471 of LNCS., Springer (2014)
10. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL, ACM (2015)
11. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009)
12. Besson, F., Jensen, T.P., Pichardie, D., Turpin, T.: Certified result checking for polyhedral analysis of bytecode programs. In: TGC. (2010) 253–267
13. Back, R.J., von Wright, J.: Refinement calculus - a systematic introduction. Graduate texts in computer science. Springer (1999)
14. Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation **6**(3-4) (1993)
15. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. TCS **277**(1-2) (2002)
16. Spiwack, A.: Abstract interpretation as anti-refinement. CoRR **abs/1310.4283** (2013)
17. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8) (1975) 453–457
18. Wadler, P.: Monads for functional programming. In: AFP. Volume 925 of LNCS., Springer-Verlag (1995)
19. Boulmé, S., Maréchal, A.: A refinement calculus to certify impure abstract computations of the Verimag Polyhedra Library – documentation and Coq+OCaml sources. `http://www-verimag.imag.fr/~boulme/vpl201503` (march 2015)
20. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP'05. Volume 3444 of LNCS. (2005)
21. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in Coq. In: TPHOL. (2005) 98–113
22. Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: ESOP. Volume 1058., Springer (1996) 219–234
23. Braibant, T., Pous, D.: Deciding Kleene Algebras in Coq. Logical Methods in Computer Science **8**(1) (2012)
24. Boulmé, S.: Intuitionistic refinement calculus. In: TLCA. (2007)

A A lightweight refinement calculus in Coq

Our implementation in Coq reformulates Section 2 with a more computational representation of binary relations. Following this idea, Appendix A.1 and Appendix A.2 present these representation changes. At last, Appendix A.3 present our datatypes for correctness diagrams of abstract computations. Appendices A.1 and A.3 also detail how the framework is adapted in order to handle alarms during the analysis.

A.1 Representations of abstract computations

A relation R of $\mathcal{R}(A, B)$ can be equivalently seen as the function of $A \to \mathcal{P}(B)$ given by $\lambda x, \{y \mid x \stackrel{R}{\to} y\}$. This curryfied representation is the basis of our representations for abstract computations. Indeed, we need to provide a Coq representation of $\mathcal{R}(^\sharp D, ^\sharp D)$ that can be turned into an OCaml type $^\sharp D \to {}^\sharp D$ at extraction. This is achieved by axiomatizing in Coq the type "$\mathcal{P}(^\sharp D)$" as "$?^\sharp D$" where "$?.$" is the type transformer of may-return monads introduced in [9] and recalled below. More generally, impure abstract computations of $\mathcal{R}(A, B)$ in Figure 4 are actually expressed in our Coq development as functions of $A \to ?B$ in a given may-return monad. Indeed, the interface of may-return monads also allows to hide data-structure details – such that handling of alarms – for the correctness proof of abstract computations. The next paragraphs detail these ideas.

Definition of may-return monads. For any type A, type $?A$ represents impure computations returning values of type A. Type transformer "$?.$" is equipped with a monad [18]:

- Operator $\gg\!=_{A,B}\colon ?A \to (A \to ?B) \to ?B$ encodes OCaml "$\mathbf{let}\, x = k_1 \,\mathbf{in}\, k_2$" as "$k_1 \gg\!= \lambda x, k_2$".
- Operator $\Uparrow_A\colon A \to ?A$ lifts a pure computation as an impure one.
- Relation $\equiv_A\colon ?A \to ?A \to \text{Prop}$ is a congruence (*w.r.t.* $\gg\!=$) which represents equivalence of semantics between impure computations. Moreover, operator $\gg\!=$ is associative and admits $\Uparrow$ as neutral element.

Last, $?A$ is equipped with a relation $\leadsto_A\colon ?A \to A \to \text{Prop}$ and we write "$k \leadsto a$" to denote the property that "computation k may return a". This relation must be compatible with $\equiv_A$ and satisfies the following axioms:

$$\Uparrow a_1 \leadsto a_2 \;\Rightarrow\; a_1 = a_2 \qquad\qquad k_1 \gg\!= k_2 \leadsto b \;\Rightarrow\; \exists a, k_1 \leadsto a \wedge k_2\, a \leadsto b$$

Impure computations and may-return monads. Abstraction of "$\mathcal{P}(A)$" as type "$?A$" is detailed on Figure 6. Conversely, for any may-return monad, a computation k of $A \to ?B$ represents a relation of $\mathcal{R}(A, B)$ defined by $d \stackrel{k}{\to} d' \triangleq k\, d \leadsto d'$.

VPL is parametrized by a *core* may-return monad which axiomatizes extern computations. This monad avoids a potential unsoundness by expressing that

$$?A \triangleq \mathcal{P}(A) \qquad k_1 \equiv k_2 \triangleq \forall x, x \in k_1 \Leftrightarrow x \in k_2 \qquad k \leadsto a \triangleq a \in k \qquad \Uparrow a \triangleq \{a\}$$

$$k_1 \gg\!= k_2 \triangleq \bigcup_{a \in k_1} (k_2\, a)$$

Fig. 6. Predicate instance of may-return monads

$${}^{c}?A \triangleq A \qquad k_1 \,{}^{c}\!\!\equiv k_2 \triangleq k_1 = k_2 \qquad k \,{}^{c}\!\!\leadsto a \triangleq k = a \qquad {}^{c}\!\!\Uparrow a \triangleq a \qquad k_1 \,{}^{c}\!\!\gg\!= k_2 \triangleq k_2\, k_1$$

Fig. 7. Identity implementation of the core monad

extern oracles are not pure functions, but encode relations. It is instantiated at extraction by providing the identity implementation given on Figure 7. Of course, the implementation of the core monad remains hidden for our Coq proofs: they are thus valid for any instance of a may-return monad.

Alarm handling in the analyzer. Our toy analyzer, specified on Figure 3, handles alarms in the style of Verasco. On a potential error, it does not stop its analysis, but writes an alarm – represented here as a value of type `alarm` – and continues the analysis. Technically, this corresponds to lift the core monad through a *writer monad transformer* [22]. Actually, we assume that the core monad has already an operation to write alarms ^{c}`write` : `alarm` $\rightarrow$ ${}^{c}?$`unit` which is efficiently extracted as OCaml extern code. Our *alarm writer monad* thus only encodes the underlying list of alarms as a boolean: `true` corresponds to an empty list of alarms. It is defined as Figure 8 where alarm writer (resp. core) constructs are prefixed by a "^{w}" (resp. "^{c}"). The implementation of ${}^{w}\!\!\leadsto$ means that the formal correctness of abstract computations with at least one alarm holds trivially. Hence, on a ${}^{\dagger}\mathbb{K}$ diagram, an abstract computation diverges (*i.e.* produces no result) as soon as it produces an alarm, whereas in the actual implementation, it produces a result which may be used to find more alarms (without formal guarantee on their meaning).

Figure 8 also defines operator $\texttt{lift}_A$: ${}^{c}?A \rightarrow {}^{w}?A$. Using `lift`, it is straightforward to lift VPL abstract domains with computations in the core monad to abstract domains with computations in the alarm writer monad. At last, operator ${}^{w}\texttt{write}_A$: `alarm` $\rightarrow A \rightarrow {}^{w}?A$, such that ^{w}`write` $m\, a$ writes alarm m and returns value a, is invoked in the implementation of ${}^{\dagger}\mathbb{K}$ assert command.

In summary, alarm writer monad instantiates our notion of analyzer correctness into "*if the analyzer terminates without raising any alarm, then the analyzed program has no runtime error*". Thanks to our compositional design through monads, reasonings on alarm handling appear only in the implementation of the alarm writer monad. Indeed, for ${}^{\dagger}\mathbb{K}$ diagrams, raising an alarm is a particular case of failure (or, more formally, non-termination) in the analysis.

$$^{w}?A \triangleq {}^{c}?(A \times \texttt{bool}) \qquad k_1 \mathbin{^{w}\!\!\equiv} k_2 \triangleq k_1 \mathbin{^{c}\!\!\equiv} k_2 \qquad k \mathbin{^{w}\!\!\leadsto} a \triangleq k \mathbin{^{c}\!\!\leadsto} (a, \texttt{true})$$

$$^{w}\!\Uparrow\! a \triangleq {}^{c}\!\Uparrow\!(a, \texttt{true}) \qquad k_1 \mathbin{^{w}\!\!\gg\!=} k_2 \triangleq k_1 \mathbin{^{c}\!\!\gg\!=} \lambda(a_1, l_1), (k_2\, a_1) \mathbin{^{c}\!\!\gg\!=} \lambda(a_2, l_2), {}^{c}\!\Uparrow\!(a_2, l_1 \wedge l_2)$$

$$\texttt{lift}\, k \triangleq k \mathbin{^{c}\!\!\gg\!=} \lambda a,\ {}^{c}\!\Uparrow\!(a, \texttt{true}) \qquad {}^{w}\texttt{write}\, m\, a \triangleq {}^{c}\texttt{write}\, m \mathbin{^{c}\!\!\gg\!=} \lambda\_,\ {}^{c}\!\Uparrow\!(a, \texttt{false})$$

Fig. 8. Alarm writer monad and its specific operators

A.2 Representation of concrete computations

We consider the issue to mechanize refinement proofs of $\mathbb{K}$ computations. Definition of $\mathbb{K}$ in Section 2.1 uses operators inspired from regular expressions. Formally, $\mathbb{K}$ embeds the Kleene algebra[3] of $\mathcal{R}(D, D)$: if K_1 and K_2 are in $\mathcal{R}(D, D)$, then $K_1 \,;K_2 = K_1 \cdot K_2$. However, $\mathbb{K}$ do not satisfies itself all properties of a Kleene algebra. In particular, " ;" has two distinct left-zeros $\perp$ and $\lightning$. Thus, it has no right-zero. We can thus not apply directly existing Coq tactics for Kleene algebras[23].

Like in standard refinement calculus [13], we simplify refinement proofs by computations of weakest-preconditions [17]. More exactly, we use weakest-*liberal*-preconditions (WLP) because they appear naturally in correctness diagram of abstract computations (as this will be illustrated by Figure 11 below). Fundamentally, this comes from the fact that weakest-liberal-preconditions do not aim to ensure termination of programs – like our static analyzes – on the contrary to original weakest-preconditions of Dijkstra.

Simplifying refinement goals by WLP. Given $K \in \mathcal{R}(D, D_{\lightning})$, the WLP of K, noted here $[K]$, is a function of $\mathcal{P}(D) \to \mathcal{P}(D)$ defined by:

$$[K]P \triangleq \{d \in D \,|\, \forall d' \in D, d \xrightarrow{K} d' \Rightarrow d' \in P\}$$

The interest of WLP is to propagate function computations through sequences of relations. Indeed, WLP transforms a sequence into a function composition: $[K_1 \,; K_2]P = [K_1]([K_2]P)$. Hence, it avoids existential quantifier of relation composition defining $x \xrightarrow{K_1\,;\,K_2} z$ as $\exists y, x \xrightarrow{K_1} y \wedge y \xrightarrow{\uparrow K_2} z$, which is tedious to handle in proofs. Moreover, for f of type $D \to D$, $[\uparrow f\,]P = \{d \,|\, f(d) \in P\}$. This allows for instance to compute $[\uparrow f_1 \,; \uparrow f_2]P$ as $\{d \,|\, f_2(f_1(d)) \in P\}$.

WLP computations allow to simplify refinement proofs, because $K_1 \sqsubseteq K_2$ is equivalent to $\forall P, [K_2]P \subseteq [K_1]P$. We list below WLP of main guarded-commands:

$$\left[\vdash P'\right] P = P' \cap P \qquad \left[\dashv P'\right] P = (D \setminus P') \cup P$$

$$\left[\bigsqcup_{a \in A} K_a\right] P = \bigcap_{a \in A} [K_a]P \qquad \left[\bigsqcap_{a \in A} K_a\right] P = \bigcup_{a \in A} [K_a]P$$

[3] A Kleene algebra is an idempotent (and thus partially ordered) semiring endowed with a closure operator. It generalizes the operations known from regular expressions: the set of regular expressions over an alphabet is a *free* Kleene algebra.

```
Record ℙ(A:Type) := {
  app:> (A → Prop) → Prop;
  app_monot(P Q:A → Prop): app P → (∀ d, P d → Q d) → app Q}.
```

$$k_1 \overset{\mathbb{P}}{\sqsubseteq} k_2 \triangleq \forall P, (k_2\, P) \rightarrow (k_1\, P) \qquad \overset{\mathbb{P}}{\Uparrow} a \triangleq \{\mathtt{app} := \lambda P, (P\, a)\}$$

$$k_1 \overset{\mathbb{P}}{\gg=} k_2 \triangleq \{\mathtt{app} := \lambda P, (k_1\, \lambda a, (k_2\, a\, P))\} \qquad \overset{A}{\sqcap} \triangleq \{\mathtt{app} := \lambda P, \exists a\!:\!A, (P\, a)\}$$

Fig. 9. Coq definitions for main operators of monad $\mathbb{P}$

In the following, we use a fundamental property of $[K]$: it is a *monotone predicate transformer*. This means that if $P_1 \subseteq P_2$ then $[K]P_1 \subseteq [K]P_2$.

A shallow embedding of WLP computations. In the style of [24], we use a shallow embedding of WLP computations. This means that we avoid to introduce abstract syntax trees for $\mathbb{K}$ computations, which would induce many difficulties because of binders in $\bigsqcup$ and $\bigsqcap$ operators. Instead, we represent $\mathbb{K}$ computations directly as monotone predicate transformers. In other words, our syntax for $\mathbb{K}$ guarded commands is directly provided by a given set of Coq operators on monotone predicate transformers (corresponding to some WLP computations).

Actually, by exploiting type isomorphism $\mathcal{P}(D) \rightarrow \mathcal{P}(D) \simeq D \rightarrow \mathcal{P}(\mathcal{P}(D))$, we encode monotone predicate transformers as functions $D \rightarrow \mathbb{P}(D)$ where $\mathbb{P}$ is the monad of *monotone predicates of predicate*. Indeed, monotone predicates of predicate are simpler and more general than monotone predicate transformers. In particular, all composition operators of predicate transformers can be derived by combining only *atomic* operators with the $\gg=$ operator of monad $\mathbb{P}$. We illustrate this point on Figure 10: A-indexed meet operator of $\mathbb{K}$ is derived from atomic operator $\overset{A}{\sqcap}$ of $\mathbb{P}$.

Figure 9 sketches the Coq definitions of this monad. An element of type $(\mathbb{P}\, A)$ is a record with two fields: a field `app` representing a predicate of $\mathcal{P}(\mathcal{P}(A))$, and a field `app_monot` which is a proof that `app` is monotone. Here, elements of $(\mathbb{P}\, A)$ are implicitly coerced into functions through field `app`. In this figure, each record definition generates a proof obligation for the missing field `app_monot`.

A lightweight formalization of $\mathbb{K}$ *in* Coq. Figure 10 illustrates how we derive guarded-commands of $\mathbb{K}$ from operators of $\mathbb{P}$ monad. We derive in a similar way operators for unbounded join (operator $\sqcup$), binary join and meet, assume (operator $\dashv$) and assert (operator $\vdash$).

With this representation change, a relation Q in $\mathcal{R}(D, D)$ is now embedded in $\mathbb{K}$ as $\overline{Q} \triangleq \bigsqcup_{d' \in D} \dashv \{d \mid d \overset{Q}{\rightarrow} d'\}\,; d'$. We can thus still express Hoare specifications (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ by $\vdash P; \overline{Q}$. Hence, we express unbounded iteration by a meet over inductive invariants as explained in Section 2.1.

On the contrary to [24], we do not proved in Coq the properties of $\mathbb{K}$ algebra. On refinement goals, we let Coq computes weakest-preconditions and

$$\mathbb{K} \triangleq D \to \mathbb{P}\, D \qquad K_1 \sqsubseteq K_2 \triangleq \forall d, (K_1\, d) \,{}^{\mathbb{P}}\!\!\sqsubseteq (K_2\, d) \qquad \uparrow\! f \triangleq \lambda d, {}^{\mathbb{P}}\!\Uparrow\!(f\, d)$$

$$K_1\, ; K_2 \triangleq \lambda d, (K_1\, d) \,{}^{\mathbb{P}}\!\!\gg\!= K_2 \qquad \prod_{a:A} K_a \triangleq \lambda d, \overset{A}{\sqcap} \,{}^{\mathbb{P}}\!\!\gg\!= \lambda a\!:\!A, (K_a\, d)$$

Fig. 10. Coq definitions for main $\mathbb{K}$ operators

```
Record †𝕂: Type := {
  impl:♯D →?♯D; spec:𝕂;
  impl_correct: ∀♯d♯d′, (impl ♯d) ⇝ ♯d′ → ∀d, d ∈ γ(♯d) → (spec d γ(♯d′)) }.
```

Fig. 11. Sketch of the Coq definition for $^{\dagger}\mathbb{K}$ datatype

simply solve the remaining goal with standard Coq tactics. This gives us well-automated proof scripts in practice. Thus, Coq code for $\mathbb{K}$ operators (with $\mathbb{P}$ included) remains very small (around 150 lines, proofs and comments included).

A.3 Representations of correctness diagrams

The Coq definition of $^{\dagger}\mathbb{K}$ datatype, sketched in Figure 11, is actually parametrized by a structure of may-return monad: abstract computations are functions of $^{\sharp}D \to ?^{\sharp}D$. Here, $^{\sharp}D$ equipped with its operators (satisfying the interface given at Figure 4) is also a parameter of the definition. Hence, our modular design allows to have abstract computations that do handle alarms, like in our toy analyzer, or that do not, like in our linearization procedure. Indeed, in abstract interpreters, detection of runtime errors (and handling of alarm) is generally done at the top-level interpreter of the analyzer, but not in the internal levels. Our notion of diagram can handle both cases in a generic way.

Hence, Figure 11 defines values of $^{\dagger}\mathbb{K}$ as triples with a field `impl` being an abstract computation, a field `spec` being a concrete computation and a field `impl_correct` being a proof that `impl` is correct w.r.t `spec`. Such proofs are simplified by applying together the WLP embedded in `spec` and the WLP already designed by [9] which simplifies reasonings with $\rightsquigarrow$ relation.

At last, `impl` being the only informative field of $^{\dagger}\mathbb{K}$ record, type $^{\dagger}\mathbb{K}$ is exactly extracted as type $^{\sharp}D \to ?^{\sharp}D$. And a $^{\dagger}\mathbb{K}$ command is exactly extracted in Ocaml as its underlying abstract computation. Here again, the Coq code for $^{\dagger}\mathbb{K}$ operators (diagrammatic proofs included) is small (around 200 lines, without the implementation of the alarm writer monad).